CISC 322

Conceptual Architecture of Google Chrome

Tom Hauser
Andrew Hoyt
Allan Read
Corey Sacrey

October 23, 2009

Table of Contents

Abstract

The following is a report on the conceptual architecture of Google Chrome. Google Chrome is an open-source web browser created by the developers at Google. Overall we have come to the conclusion that the architectural style of Google Chrome is a hybrid between layering and object-oriented. The structure itself leans closer towards layered but one specific component requires an object-oriented style. Since most of the communication between components requires the data to be sent through the IPC, the structure around the Browser component is closer to object-oriented. Almost everything returns back to the browser component before it can be sent to its proper destination.

The only component that deviates from layering is the RenderingObject. Google Chrome creates a separate process for each tab that is opened within the browser. This unique threading model causes each tab to receive its own RenderingObject. Therefore at any given point there may be many instances of the RenderingObject component within Chrome. This adds to the stability of Chrome since if one tab goes haywire and freezes it should not affect any other instance of RenderingObject. RenderingObject's main subcomponent is WebKit. This is the same rendering engine that many other web browsers use including Apple's Safari.

Another key component that sets Chrome apart from other web browsers is its powerful open-source JavaScript engine called V8[13]. Rather than use one of the current JavaScript engines Google decided to build its own. The end result was what is considered one of the most efficient JavaScript interpreters around.

Introduction

Microsoft's Internet Explorer has long been the leader in the web browser industry. By January of 2004, 84.7%[15] of internet users chose Internet Explorer (IE) 6 and IE 5 over the competition. With the release of Mozilla Firefox in November of 2004, a large number of users switched their browser usage to Firefox. In November 2006, two years after the initial release date of Firefox, usage soared to an astounding 29.9%[15]. Consequently, Internet Explorer's usage dropped to 60.6%[15] -- a decrease of almost 15% in two years.

With an ever-growing market in the web browser realm, Google decided it was time to introduce their browser. In September 2008[14], Google launched a beta version of their web browser titled Google Chrome. By early December 2008[14], a stable public version was released. Within one year of the official release of Google Chrome, usage statistics show that 7.1%[1] of internet users choose Google Chrome over the competition.

Google Chrome is undergoing continuous development as Chrome's open sourced project was released to the public entitled Chromium. It aims to achieve improved security, speed and stability. To improve security, Chrome will regularly update two "blacklists", which are aimed at improving a user's overall feeling of security by warning users when they have attempt to enter a site which is affiliated with malware or phishing. Chrome has also attempted to improve its overall speed with the design and implementation of the V8 JavaScript Engine, made specifically for Chrome[13]. Additionally, stability has been enhanced in Chrome by creating a new process for each new tab that is opened. This enhancement fixes the problem of an entire browser crashing as a result of an error in one tab. Instead, if an error has occurred in a tab, only the error-containing tab will crash, leaving the others unharmed and still operating.

The purpose of this report is to examine the conceptual architecture of Google Chrome. By having a reference architecture of a web browser available to view in the paper, *A Case Study in Architectural Analysis: The Evolution of the Modern Web Browser* (Grosskurth and Godfrey), we were able to use this as a solid base to begin the discussion of Chrome's conceptual architecture. A diagram of the architecture is included showing the relative dependencies among the components.

Additionally, we will be providing two use cases, which will give an example of important actions that a user performs while using Google Chrome. One use case diagram will demonstrate the sequence of actions that will be performed by the component subsystems in Google Chrome when a user attempts to access a webpage which is *not* stored in the cache. A second use case diagram will be a representation of when a user attempts to access a webpage that *is* stored in cache.

By analyzing the reference architecture in the aforementioned paper, we were able to begin deciding on the key aspects of Google Chrome's conceptual architecture. We determined that Chrome's conceptual architecture is comprised of eight subsystem components: ChromeViews User Interface, Browser, RenderingObject instances, Network, JavaScript V8, Plugins, Display Backend and Data Persistence. This report will also address the interactions between these subsystems and how these interactions contribute to the overall Google Chrome experience. As a group, after establishing the dependencies among the various subsystems, we came to the conclusion that Google Chrome's conceptual architecture is a hybrid of the object oriented architecture style and the layered architecture style.

Conceptual Architecture

After our investigation of Chrome's inner workings, we have decided that Google's conceptual architecture is a hybrid of the layered and object oriented styles. With the Internet becoming what it is today, Chrome needed to become similar to an operating system with respect to handling mulitple browser windows. Also, the web is full of poorly coded websites, so making a web browser that never crashes is an impossibility. Taking these desired features and facts into account, we think that Google decided it would follow a fundamentally different approach to the standard architecture of a web browser; instead of using a single process to render all web pages, Chrome would utilize the object oriented style by creating each new tab in a new process[2], with its own partition of resources. The advantage conveyed by this is that if an individual tab hangs or encounters an error, other tabs are unaffected.

One adverse effect this feature has on the overall architecture is that there are a set of Libraries that handle all inter-component communication, and they are located in the browser. With these libraries, the browser component communicates directly to the JavaScript, plug-ins, and the display back end components. In essence, the browser acts as a mediator for any given instance of the RenderingObject. This allows less resources to be used, and also avoids every other component needing to keep track of which RenderingObject is requiring it. By centralizing everything into the browser component, only the Browser needs to know which RenderingObject is requesting which service. This kind of conceptual architecture makes sense if the main goal of the program is to provide an efficient, stable environment for web browsing. Although the reference architecture provides a good basis for deciding what most other browsers do, Google’s goal was not to create another clone browser; they wanted to revolutionize the way browsers function. This explains the great difference between the reference architecture and the conceptual architecture we have come up with.

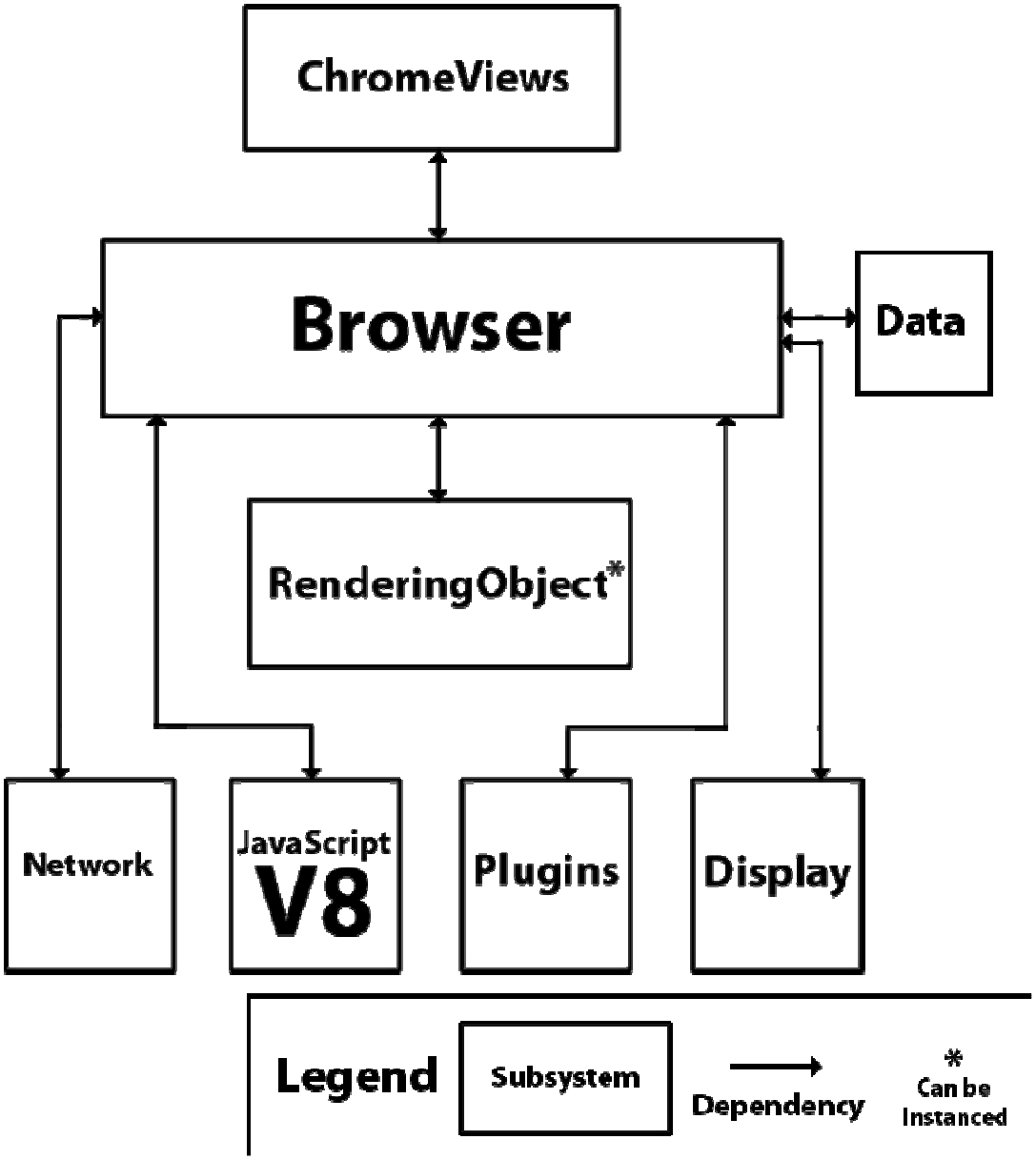

Figure 1: Conceptual architecture of Google Chrome

Besides this object oriented approach to RenderingObject instances, the rest of the architecture is designed in a semi - strict layered style. The UI component takes input from the user and is the frame in which the RenderingObject is displayed. The browser component is the real workhorse of the architecture. It manages all inter-component communication as well as managing all of the open RenderingObject instances. The JavaScript component parses any JavaScript content on a given web page, sending the contents back to the RenderingObject through the browser. The browser is also responsible for all network accesses that a particular RenderingObject instance may need. The Data component is responsible for caching, cookies, bookmarks, and all other information that the browser needs to remain on the hard disk.

Description of Components

User Interface

The User interface in Chrome is built upon the native Windows framework, and includes another component called views[6]. Views is the component that is responsible for all organizational information in the Chrome UI. When it has finished its organization, it makes calls, through the browser component, to the Display libraries, of which there are three: Skia, GDI, and NativeControls. The Skia component is used to render the vast majority of the User Interface, and is also a Google in house library. The reasons for implementing this library in Chrome can be found in the design documents. Chrome's user interface is too different to utilize Windows libraries exclusively. Some features, such as the Tabs in the title bar, cannot be done without external libraries. Therefore, Chrome uses Skia for most of its UI. GDI is a component that is used for Text Rendering. It is used because it gives Chrome a more native to windows look, as opposed to having Skia do the text. The NativeControls component is only responsible for a small amount of UI elements, such as Radio Buttons and Checkboxes. These are indicated in Views by a special kind of element, and this element calls NativeControls for the information to display.

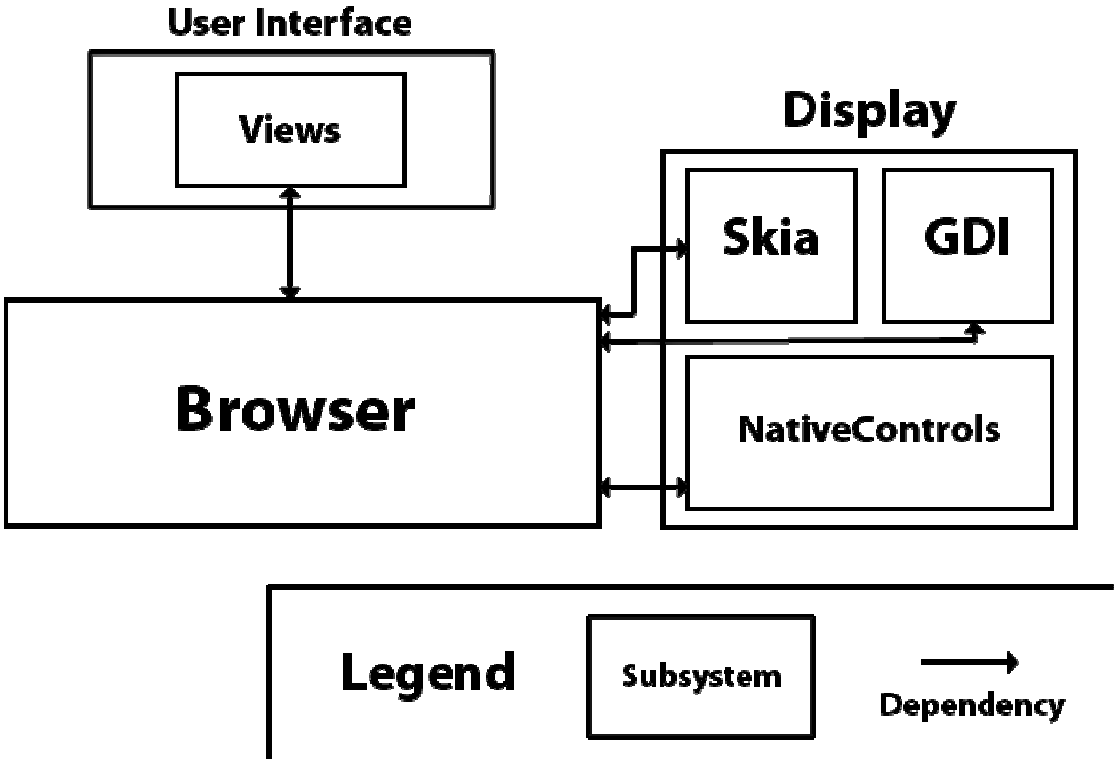

Figure 2: Breakdown of User Interface component.

Browser Component

The Browser Component (figure 3) provides a common point of entry and exit to the multiple renderer objects[2]. One of Google Chrome's major differentiators, right from the design stage, has always been its multiprocess architecture. By default, Chrome runs almost all open tabs in different processes. It accomplishes this by having one object, called the "browser", manage multiple instances of the "renderer" objects. The browser tracks all of the renderer object instances via multiple instantiations of the Render Process Host (RPH) module. Each instance of the renderer object talks directly with its associated RPH inside the browser component, via its own IPC channel.

However, by default, not all tabs necessarily get their own process. In order to cut down on the amount of system resources per rendered page, Chrome includes the ability to have multiple pages share the same renderer object[10]. By default, this only occurs if chrome considers two pages "related" (for example, if a JavaScript window opens up), or if Chrome has hit its process limit. In order to keep track of pages that share the same renderer object, each RPH will keep track of multiple RenderViewHosts, which will actually be responsible for the sending and receiving of messages on behalf of each rendered page.

The Resource Dispatcher Host (RDH) is what provides the link between the browser components and the other subcomponents of Chrome, other than the rendering objects[10]. The RDH is what receives the data from the network stack that is to be passed down to the renderer. It is also what decides what information needs to be passed to the data persistence (things that are to be stored in cache, etc.).

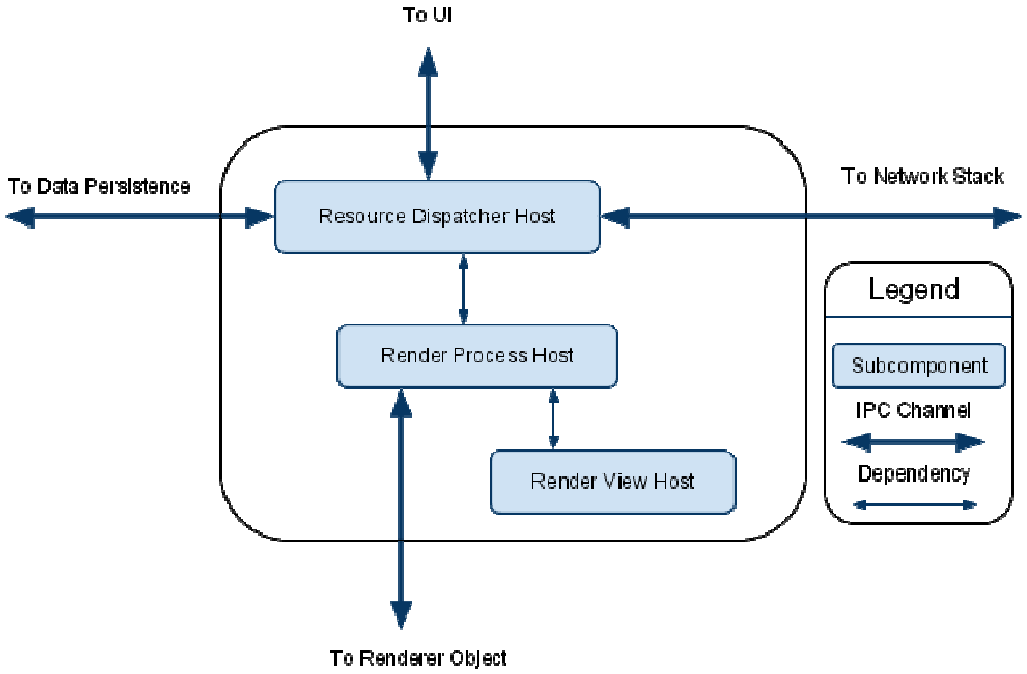

Figure 3: Breakdown of the Browser Component

Renderer Object

The Renderer Object (figure 4) is responsible for creating the bitmap that is eventually displayed on the screen[2]. In most cases, each Chrome tab will have its own renderer object. Inside each renderer object is what Google calls a "renderer process" which contains all of the logic required for maintaining multiple "render views". This allows Chrome the functionality, mentioned earlier, to combine multiple tabs inside one render object as necessary. The render views themselves are what store the current state of the displayed page as well as the logic required to update it. It is these render views that talk with WebKit, Google chrome's rendering engine. Each renderer object has its own instantiation of the WebKit library. While perhaps more resource intensive, this is what provides chrome with its ability to sandbox each web page. This guarantees that, even if one page is having trouble rendering, every other page will still be able to render correctly and speedily. This way each instance of WebKit (the most likely cause of crashes) is safely insulated from the host machine as well as the other Chrome components inside its renderer object.

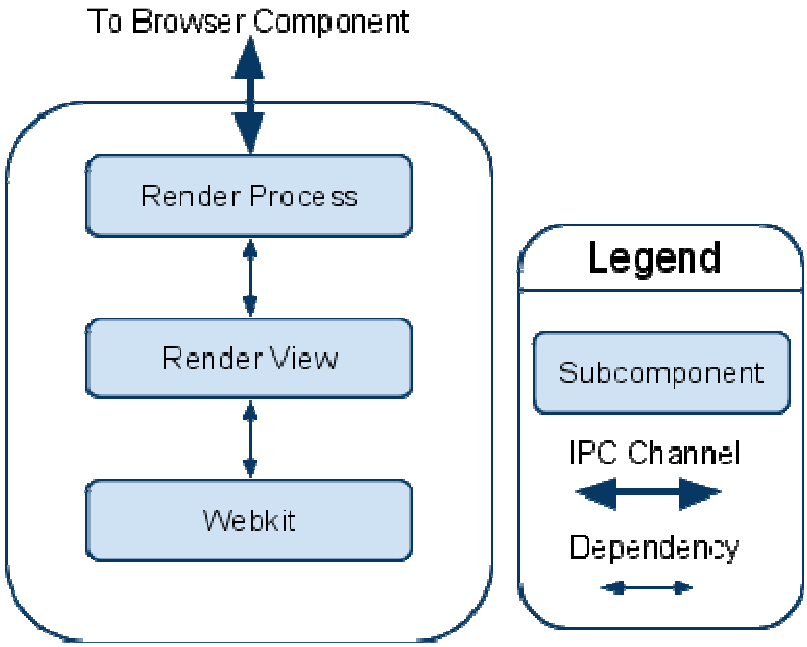

Figure 4: Breakdown of Renderer Object

Networking

The networking component of Google Chrome is the component that enables the web browser to connect to servers in order to receive data, most likely in the form of web pages. This component currently supports two Windows exclusive libraries, WinHTTP and legacy support for WinInet[8]. With Mac OS X and Linux versions of Chrome in beta, the Chromium team is in the process of transitioning to a re-written networking component that would be fully portable to these new platforms. The basic code has been developed and is in the most recent beta releases of the Google Chrome browser. Some work is still to be done though as SSH support still relies on these legacy Windows-exclusive libraries.

JavaScript V8 Engine

Google Chrome uses the JavaScript V8 engine (written in C++)[13], which is an open source engine originally developed by Google in Denmark, specifically for Google Chrome. The JavaScript V8 engine talks directly with the Browser component. It provides the parsing of JavaScript, as well as a virtual environment to allow the JavaScript to be executed.

The V8 engine's increased performance is attributed to the fact that it compiles all JavaScript directly to machine code instead of either interpreting it or converting it to bytecode. Another important aspect to V8's performance is its very efficient garbage collector.

Data Persistence

The Data Persistence component talks directly with the Browser component. This component of the Google Chrome Conceptual Architecture is used to store data associated with the user's browsing experience. This includes data such as cookies, browsing history, favourites and cache. By talking with the Browser component and not the individual RenderingObject component(s), it allows the data to stay constant throughout Google Chrome. Each tab must be able to access the same persistent information.

Display Backend

The Display Backend component talks directly with the Browser component. It provides the main drawing and windowing essentials, as well as the necessary fonts, in order to properly display the webpage on the User Interface (via the Browser). There are three libraries within the Display Backend -- Skia, Graphics Device Interface (GDI) and a NativeControls subcomponent. Skia is responsible for drawing the 2-dimensional text, geometries and images[12]. In Chrome, GDI is mainly used for text rendering. The NativeControls subcomponent is very basic in the sense that it provides fairly little for the User Interface.

Plug-Ins

The plug-in component of Google Chrome is a very key component. Recently the ability to add new functionality in the way of third-party plug-ins has become almost a required element of web browsers. However, just giving plug-ins full reign is not the best course of action. One of the key aspects of Chrome is trying to create a very stable browser where there can be multiple processes and if plug-ins were spread across these processes then it would defeat one of Google's main goals. Therefore, this leads to each plug-in having their own process just like every tab[5]. If one plug-in freezes then it should not affect the other plug-ins or browsers. These plug-in processes are managed by the plug-in component which sends and receives commands via the IPC channel[4].

Use Cases

View Webpage Not in Cache

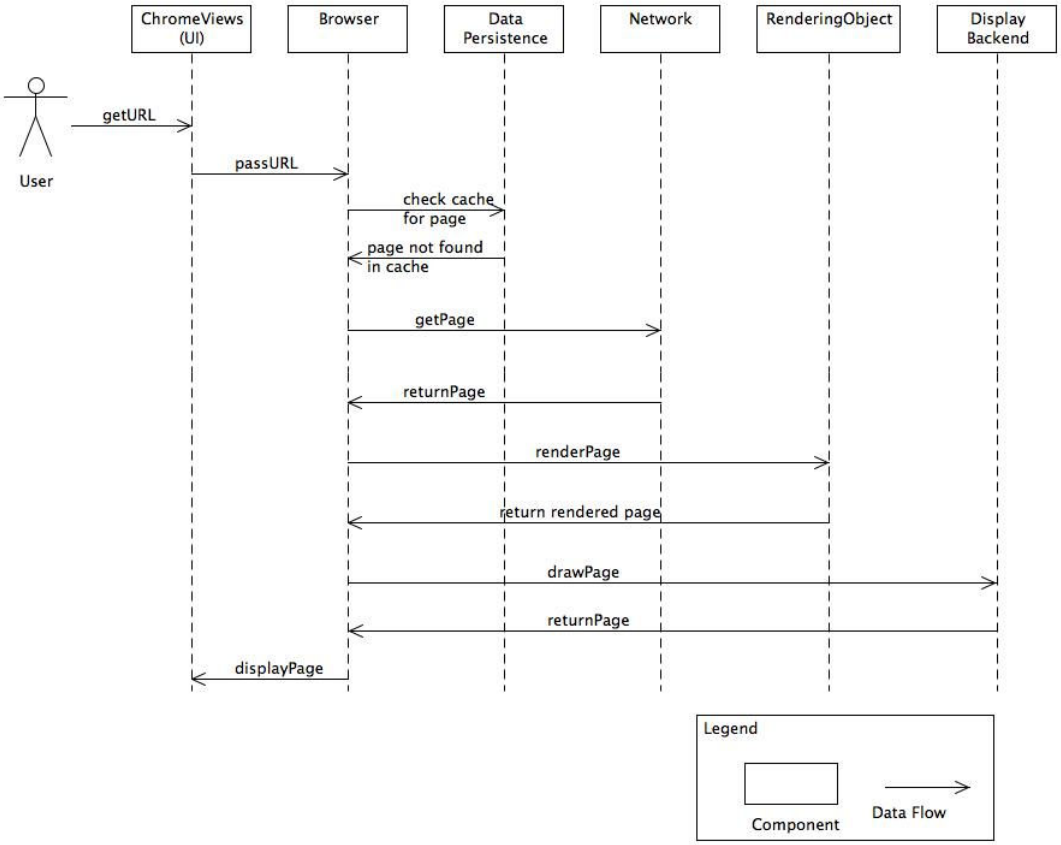

Figure 5: Sequence Diagram for Use Case of a web page not in cache

This sequence diagram is displaying the necessary steps that Google Chrome progresses through starting when the user enters a website address into the 'Omnibox' located at the top of the user interface. Next, the URL is passed to the Browser component. The website the user has entered is then checked in the Data Persistence component if it exists in the cache. In this case, the website is not found in the cache and control is given back to the Browser component.

The Browser then passes on control to the Network component, which subsequently identifies which type of protocol to use -- in this case being hypertext transfer protocol (HTTP) since the user has entered a webpage to view. The Network component then determines which web server to contact by converting the website address to an IP address. The located webpage information is returned to the Browser.

A RenderingObject specifically for this webpage will render the page as it is received from the Browser. The Display Backend component will then provide the drawing and windowing

essentials and necessary fonts needed for the proper display of the webpage. The page will then be returned back to the Browser, and next displayed to the user in the ChromeViews User Interface.

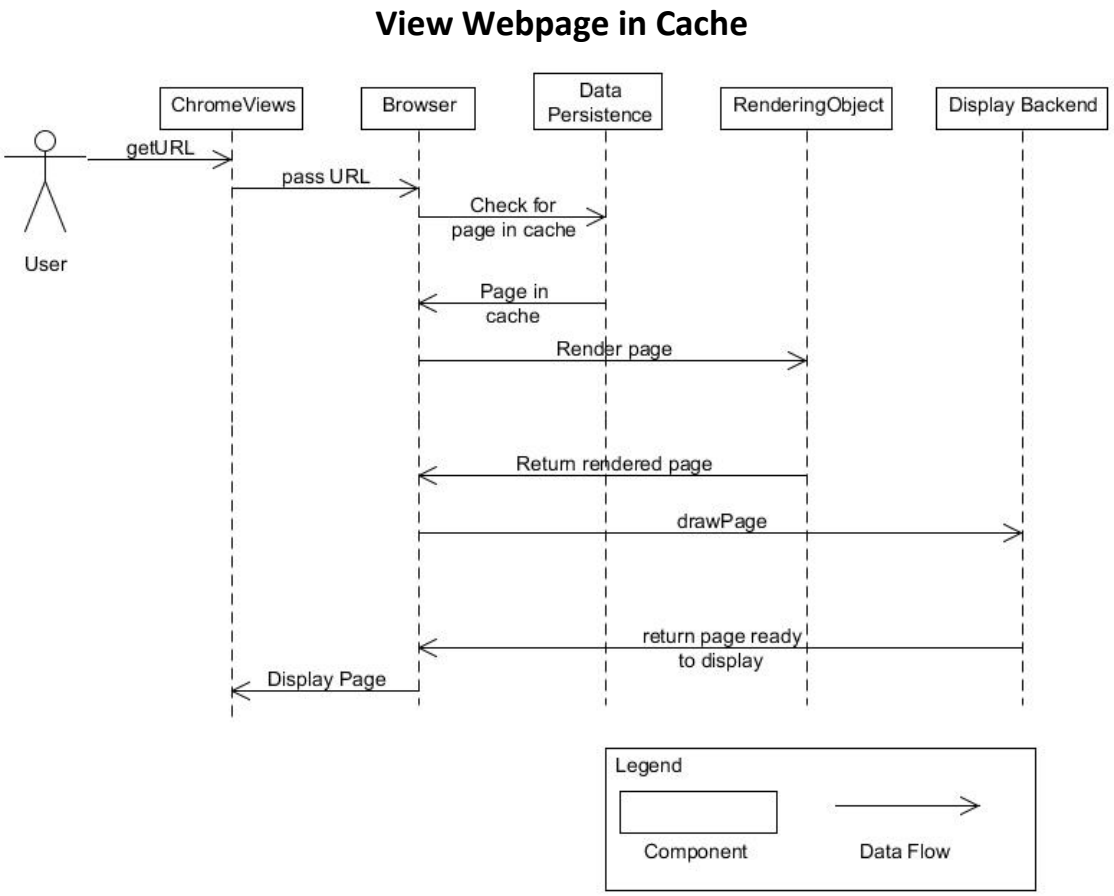

Figure 6: Sequence Diagram for Use Case of a web page in cache

This is a relatively simple use case of the user requesting to view a page that already exists in cache. The user will enter the URL of the site he/she wants to visit. ChromeViews will then pass this URL on to the browser component. The browser component will then check data persistence to see if the page exists in cache. In this example, the page does exist in cache. The data will be returned to the browser who will then send it to the RenderingObject to render. WebKit will do the majority of this rendering inside the RenderingObject component. The page is then sent to the Display Backend to set up the required fonts and other visual requirements. Finally the page will be sent to the UI to be displayed on the screen.

Lessons Learned

Overall we had a little bit of difficulty initially with our researching. We focused too much on trying to find every part in Google Chrome that matches a component in the web browser reference architecture rather than researching how Chrome worked. Google Chrome runs vastly different than most other web browsers due to its unique threading system where each tab and plug-in gets its own process. If we had a second opportunity we probably would have focused less on the reference architecture and just used it as a guide rather than trying to build around it.

The official Chromium developer documentation was a great starting point but it did not include all the information that was necessary to figure out the key components. Due to the nature of open-source software, the developer documentation sometimes does not include the exact information that we needed. The documentation was lacking in explanation that allowed someone who had no knowledge of Chrome initially, to jump in and be able to understand how the browser operated internally. Therefore we had to look elsewhere for sources of information that brought the information into broader terms that made it easier to understand

Another problem we faced was the lack of information on some key components inside Chrome, especially the data persistence and display backend components. The developer documents do not include much information on these components other then knowledge that they exist. Other sources do not go into much detail on how Chrome implements data persistence or the display backend.

Some components we discussed at great length about whether they should be in the architecture or whether they would just be a part of one of the current subsystems. One such example was WebKit, which is a part of RenderingObject. WebKit is a key component in the rendering of web pages. It does reside as a part of the RenderingObject though. Eventually we decided to leave most of the smaller components out of the conceptual architecture. This allowed us to leave it as general as we could while still providing enough information to understand the architectural style of Chrome. These smaller components are mentioned in the descriptions of the subsystem they belonged to.

Conclusion

After examining the Google Chrome Developer Documentation, our group settled on the conceptual architecture being a hybrid of two styles -- layered and object oriented. The layered style encompasses most of the architecture. The User Interface talks to the layer below, the Browser, which subsequently talks to the bottom layer being composed of the Network, JavaScript V8 Engine, Plugins and the Display Backend. The object oriented style is apparent in the conceptual architecture as each instance of the RenderingObject is able to talk back and forth between the Browser component. Additionally, the object oriented style is manifested in the interactions between the Plugins component and the multiple instances of the RenderingObject.

Throughout the process of examining Google Chrome's architecture, lessons were learned which will help us progress further into the assignments to follow. First, we had some difficulty distinguishing between the ideas of a Conceptual Architecture and a Concrete Architecture. After consulting the Google Developer Documentation of Chrome, we had difficulty conceptualizing the architecture as the documentation is concretely defined. Additionally, we had trouble finding significant information on some of the subsystems, such as the Display Backend component.

After looking at a brief overview of Google Chrome, we were able to understand the fundamental ideas which support its ever-growing community. As a group, we will continue this investigation into Google Chrome's Concrete Architecture.

Data Dictionary

Note: All definitions from Wikipedia unless otherwise stated

Chromium - Name of the open-source project of which Google Chrome is based.

GDI - Graphics Device Interface. Fundamental operating system component. Responsible for representing graphics on displays, as well as outputting them to output devices such as printers.

Google Chrome - Web browser released by Google based off of the Chromium project. Uses WebKit as its rendering engine.

HTTP - HyperText Transfer Protocol. Protocol used to retrieve inter-linked resources, most commonly used to transfer web pages.

IP - Internet Protocol. Used as a means of communicating data over a network.

IPC - Inter-Process Communication. A way of communication among multiple threads in one or more processes.

Omnibox - Address bar of Google Chrome. Allows user to enter in a web address to visit a web page or enter in search terms to automatically search using the Google search engine.

Open-Source - Approach to software development where the source code of a project is accessible to everyone.

SSH - Secure Shell. Network protocol that allows encrypted data to be sent via two networked machines. More secure then HTTP and is commonly used when handling passwords or other confidential information.

UI - User Interface. An interface allowing the user to communicate with the system. Provides the user with means of input and output.

WebKit - Rendering engine behind Google Chrome. Originally developed for Apple's Safari browser, it has since been ported to many platforms.

WinInet - Windows exclusing protocol handler for HTTP, HTTPS(secure HTTP) and FTP (file transfer protocol).

WinHTTP - Provides developers with a high level application programming interface to the HTTP/1.1 Internet protocol. Primarily used in server-based scenarios[11].

References

Google Chrome Developer Documents:

1. http://dev.chromium.org/developers/design-documents
2. http://dev.chromium.org/developers/design-documents/multi-process-architecture
3. http://dev.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome
4. http://dev.chromium.org/developers/design-documents/inter-process-communication
5. http://dev.chromium.org/developers/design-documents/plugin-architecture
6. http://dev.chromium.org/developers/design-documents/chromeviews
7. http://dev.chromium.org/developers/design-documents/graphics-and-skia
8. http://dev.chromium.org/developers/design-documents/network-stack
9. http://dev.chromium.org/developers/design-documents/disk-cache
10. http://dev.chromium.org/developers/design-documents/multi-process-resource-loading

Microsoft's "About WinHTTP":

11. http://msdn.microsoft.com/en-us/library/aa382925(VS.85).aspx

Description of skia:

12. http://code.google.com/p/skia/

Description of V8:

13. http://code.google.com/apis/v8/

Wikipedia Entry on Google Chrome:

14. http://en.wikipedia.org/wiki/Google_chrome

Statistics on Most Widely Used Browsers:

15. http://www.w3schools.com/browsers/browsers_stats.asp